PSis - Programação de Sistemas 2014/2015
Exame 1ª Época, 8 de Junho de 2015, 8h00, Duração: 2h

Software Design

1. [1 value] One of the key principles of systems' design is the **Principle of Least Knowledge**.

- Describe this principle
- Explaining the problems that arise by not following it.
- Explain how this principle was applied during the development of the project.

This principles states that a component or object should not know about internal details of other components or objects. If this principle is not followed changes in the internal implementation of a component will require changes on the code of other components.
In the project when defining the server interface using protocol buffers all the server implementation was hidden from the client (event the programming language and strings encoding is hidden).

Software Architectures

2. [1 value] Peer-to-Peer architectures are widely used in systems with nodes scattered in the Internet (e.g. file sharing), but are also used in the context of centralized datacenters and clusters.

- Describe the Peer-to-Peer architecture pattern.
- Explain why this kind of architecture is fundamental to the correct functioning of large clusters such as those in Google or Facebook.

In a peer-to-peer architecture all nodes can act as server and clients. Added to this usually the management of functionalities and data is done in a decentralized distributed manner.
In the case of large cluster or datacenter the centralized management is impratical. In this case the use of protocols and architectures similar to P2P is the most efficient solution.

Software Testing

3. [1 value] The mathematical expression to convert temperatures from Fahrenheit to Centigrade is

T_C = (T_F - 32) * 5/9

The conversion is to be implemented as a C function: `int fahrToCent(int T_F);`

- Define and describe a set of unitary tests that will allow verifying the corresponding code. It is not necessary to write the code but you have to describe what is to be tested and how.

The test suit should include tests

- that verify common cases (100.C / and 0.C)
- that verify if the 5/9 division is correct (it may produce 0, instead of 0.555555)
- that verify if the rounding is correct
- verify with large (positive and negative) values

4. [1 value] Regression testing is one of the available test classes.

- Describe the uses of regression testing and when it should be applied.

Regression verifies that changes in the code (modification on the implementation of a module, bug corrections or new functionalities) do not break existing code.
Regression tests should be done whenever new code, functionalities are introduced and is composed of the re-execution of tests that the programmer knows that have passed.

Multiprogramming

5. [1 value] Some classes of applications gain from using multithreading even if the target computer only

has one processor/core.

- Describe one of such class of applications (its characteristics and types of instructions).
- Describe why such multithreaded applications gain from executing on a single processor/core computer.

Applications where there is communication with peripheral (file system, network or keyboard) can take advantage of multithreading.
During the periods where the one thread is waiting for data (from the peripherals) another thread can be doing other processing.

6. [1 value] The following code should create 10 threads and assign each one a different value (from 0 to 9), which is then processed by the thread:

```
1.  int main(){                          10.  void * thread_code(void * arg){
2.  ...                                  11.    int n = * arg
3.    for(i = 0; i < 10; i++){           12.
4.              pthread_create(          13.    ...
1.              &threads_id[i],          14.
2.              NULL,                    15.    pthread_exit(NULL);
3.              thread_code,             16.  }
4.              &i);
5.  }
```

- Explain why the previous code is incorrect and present a case where it does not work as supposed.
- Write the C code that provides a solution to the presented problem.

If the instruction on line 11 is executed before another thread is created, the program works correctly, otherwise several threads will have n with the same value. For instance:
The first thread is started on line 4.
Before the new thread executes line 11 the n is incremented (line3) and the second thread is created.
If the second thread executes line 11 its n will be 1.
If now the first thread executes line 11 it will also have n with the value 1 (instead of 0).

This would be avoided if each thread would access its own value:
int v[10]

```
for(i = 0; i < 10; i++){
   v[i] = i;
   pthread_create(&threads_id[i],NULL thread_code,&v[i])
}
```

Or

```
for(i = 0; i < 10; i++){
   aux = malloc(sizeof(int));
   *aux = i;
   pthread_create(&threads_id[i],NULL,thread_code,aux)
}
```

Inter-process communication

7. [1 value] Besides channels (pipes, fifos, sockets) and shared memory (shm_open/mmap), related process (parents and child processes) can exchange information at creation and termination.
Without resorting to message sending or shared memory explain:

- How data can be transferred between the parent and the child.
- How to transfer data from the child to the parent.

The variables from the parent process will be copied to the child, these variables can contain relevant information for the execution of the child.
When a process exists it returns a code that the parent can retrieve with the wait function.

Synchronization

8. In a robotic flight festival a group of flying robots (quad-copters) executes figures in the air. The flying robots are arranged in a circle and during the presentation at most three go into the middle (stunt area) to do some stunts.
Each quad-copter produces either blue smoke (quadA) or white smoke (quadB).
To avoid accidents only three quad-copters (a triple) can dos stunts simultaneous:

- Two quadA;
- One quadB.

All other quad-copters remain in the outer circle.
Only a triple can be in the stunt area and a triple can only enter after all of the previous three quadcopters leave the stunt area.
The quad-copters in the stunt area exit all at the same time: after all finished doing stunts.

Suppose that quad-copter communicate with a central computer that handles the synchronization and where each quad-copter has an associated thread.
The following code uses semaphores and implements the synchronization requirements for the two types of quad-copters.

The semaphores' `wait` and `signal` functions can receive the number of units to remove or add. By default the number of units is 1.

```
sem_t mutex=1, SA=0, sB=0;
int nA=0, nB=0;
```

```
void skydiveA(void){

  wait(mutex);
  if (nA && nB){
    nA--;
    nB--;
    signal(sA,2);
    signal(sB);
  } else {
     nA++;
     signal(mutex);
  }
  wait(sA);

  do_theStunts();

  barrier();
}
```

```
void skydiveB(void){

  wait(mutex);
  if (nA>=2){
    nA-=2;
    signal(sA,2);
    signal(sB);
  }else {
    nB++;
    signal(mutex);
  }
  wait(sB);

  do_theStunts();

  barrier();
  signal(mutex);
}
```

[1 valores] 8.a) What is the purpose of the `signal(mutex)` in the end of `skydiveB(void)` function?
Could the `signal(mutex)` be transferred to the end of `skydiveA(void)` (after the barrier

function)?

The instruction signal(mutex) in the end of the skydiveB is to allow the creation of new triples that will reach the center after all elements of the current triple had finished the performance. Note that the process that signals the mutex is not necessarily the same as the one that did the last wait(mutex) and gave permission to move to the center. Although this may seem strange it is lawful; one and only one element of the triple locks the mutex when the stund starts, and one and only one unlocks it in the end. If this statement was passed to skydiveA it would be executed by two processes of type A, which would put excess units in the semaphore.

[2 valores] 8.b) The `barrier()` function guarantees that all three quad-copters leave the function after they all conclude the `do_theStunts()`.
Program it using all the necessary global variables (semaphores or integers).

The standard implementation of the barrier is as follows.

```
int ctr=0;
sem_t mutexbarr=1, sembarr=0;

void barreira(void){
        esperar(mutextbarr);
        if (++ctr==3){
                ctr=0;
                assinalar(sembarr,3);
        }
        assinalar(mutextbarr);
        esperar(sembarr);
}
```

In the context of the problem in hand, this realization of the barrier has a flaw. Suppose that all the elements of a triple invoke the barrier, and the scheduler is such that they run the above code until the line signal(mutexbarr), unlocking the barrier. The quabB now continues to run, leaving the procedure and signaling the mutex at the end of skydiveB. At that time a new triple can get to the center, and if one of its elements make a short stunt and invoke the barrier before the processes of the previous trio have executed wait (sembarr), it will overcome them and leave the center. To solve this issue, you must ensure that those who unlocks the barrier only frees mutexbarr after other processes waiting for the barrier indeed cross it. The solution is the following.

```
int ctr=0;
sem_t mutexbarr=1, sembarr=0, saida=0;

void barreira(void){
        esperar(mutextbarr);
        if (++ctr==3){
                ctr=0;
                assinalar(sembarr,2);
                esperar(saida,2);
                assinalar(mutextbarr);
        } else{
                assinalar(mutextbarr);
                esperar(sembarr);
                assinalar(saida);
```

```
    }
}
```